Artificial Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Licentiate Thesis in [Research Subject - as it is in your ISP]
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden [2022]

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT XXXX:XX
ISBN XXX-XX-XXXX-XXX-X

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av licentiatexamen i [ämne/subject] [veckodag/weekday] den [dag/day] [månad/month] [år/2022] klockan [tid/time] i [sal/hall], Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

WebAssembly has become the fourth official web language. This new language allows web browsers to execute existing programs or libraries written in other languages, such as C/C++ and Rust. Apart from web browsers, WebAssembly evolves to be part of Edge-Cloud computing platforms. Despite being designed with security as a premise, it is not exempt from vulnerabilities. Our approaches deal with this fact by providing a preemptive solution with software diversification.

In this thesis, we propose an automatic approach to generate software diversification for WebAssembly programs. In addition, we provide complementary implementation for our approaches, including a generic LLVM superdiversifier that potentially extends our ideas to other programming languages. We empirically demonstrate the impact of our approach by providing Randomization and Multivariant Execution (MVE) for WebAssembly. Our results show that our approaches can provide an automated end-to-end solution for the diversification of WebAssembly programs. The main contributions of this work are:

- We highlight the lack of diversification techniques for WebAssembly through an exhaustive literature review.
- We provide the implementation of two tools, CROW and MEWE. These tools provide randomization and multivariant execution for respectively.
- We include *constant inferring* as a new code transformation to generate software diversification for WebAssembly.
- We empirically demonstrate the impact of our technique by evaluating the static and dynamic behavior of the generated diversification.

Our approaches harden observable properties commonly used to conduct attacks, such as static code analysis, execution traces, and execution time. Therefore, our approaches harden unknown and yet-unknown vulnerabilities.

Keywords: WebAssembly, Software Diversification, Automatic Software Engineering, Security

Sammanfattning

Write your Swedish summary (popular description) here...
Keywords: Keyword1, keyword2, ...

Acknowledgements

Paraphrasing a good friend of mine: the persons that contributed to this work know who they are, and I prefer to thank them personally.

Javier Cabrera-Arteaga,
Stockholm, May 2022

Contents

Part I

Thesis

01 INTRODUCTION

"Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a démouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations. "

— MR. Cox ***1893**, Cinderella: Three hundred and forty-five variants [**?**]*

The Web Consortium (W3C) standarized bytecode for the web environment with the WebAssembly (Wasm) language in 2015. Wasm allows web browsers to execute existing programs or libraries written in other languages, such as C/C++ and Rust. Beyond web environments, WebAssembly evolves to be part of Edge-Cloud computing platforms [**?** **?**]. Despite being designed for sandboxing and secure execution, it is not exempt from vulnerabilities [**?**]. For example, WebAssembly engines are vulnerable to speculative execution [**?**], and C/C++ source code vulnerabilities might be ported to Wasm binaries [**?**].

One strategy to hide such vulnerabilities is to move them in time as a preemptive solution. The goal is to make potential vulnerabilities available only in a time window. This makes potential attackers not hit what they cannot see. This strategy is usually called Moving Target Defense (MTD) [**?** **?**]. MTD for software is a collection of techniques that aim to improve the security of a system by constantly rotating its vulnerable programs from one variant to another. A program variant should be different from the original program but functionally equivalent to it. By rotating the deployment and execution between the program variants, a potential attacker needs more efforts to perform the same attack for all variants [**?**]. Thus, one premise for effectively implementing MTD for a given program is the need for the program variants.

In MTD, Software Diversification is the process of finding, creating, and deploying program variants. Usually, program variants could be found in the wild in a phenomenon called natural diversity [**?**]. In the case of WebAssembly, since it is a novel technology, there is no natural diversity. Thus, effective MTD cannot be implemented due to the lack of program variants. This work proposes to create program variants for WebAssembly artificially. Therefore, we aim to generate artificial software diversification for WebAssembly. To reach such a goal, we answer three research questions enunciated in the following.

1.1 Research questions

In this section, we present our three research questions. Our research questions are formulated by merging our publications and experiences during the creation of Software Diversification for WebAssembly.

RQ_1 **To what extent can we artifically generate program variants for WebAssembly?**
With this research question, we quantitatively assess the static differences between program variants created by our approach. We answer this question at the population level, where a program population is the collection of one original program and its generated variants. We aim to investigate the code properties that increases(or diminishes) generated diversification at population level.

RQ_2 **To what extent are the generated variants dynamically different?**
With this research question, we complement RQ_1. We aim to investigate the impact on execution traces and execution times of the generated program variants.

RQ_3 **To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?**
With this research question, we aim to investigate the impact of Software Diversification for WebAssembly in an emerging technology, Edge-Cloud computing. We evaluate the impact of a novel multivariant execution approach on real-world WebAssembly programs in a world-wide scale experiment.

1.2 Contributions

This thesis contributes through four milestones. First, as a theoretical contribution, we summarize the code transformations used to artifically generate software diversification through an exhaustive literature review. Consequently, we highlight the lack of diversification techniques for WebAssembly. Second, as a technical contribution, we provide two tools, CROW [**?**] and MEWE [**?**]. Besides, we summarize the main challenges faced during their implementation. In addition, we discuss the incorporation of *constant inferring* as a new transformation. Third, we propose a methodology to quantitatively evaluate the impact of our tools, assessing the creation of artificial software diversification for WebAssembly. Fourth and final, we empirically demonstrate the impact of our technique by evaluating the static and dynamic behavior of the generated diversification.

1.3 Publications

This work is based on the following publications:

P_1 Superoptimization of WebAssembly Bytecode [**?**]
Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Programming 2020, MoreVMs'20

P_2 CROW: Code Diversification for WebAssembly [**?**]
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
NDSS 2021, MADWeb

P_3 Multi-Variant Execution at the Edge [**?**]
Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Under review

P_4 Scalable Comparison of JavaScript V8 Bytecode Traces [**?**]
Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
SPLASH 2019, VMIL

Thesis layout

This dissertation is organized in five chapters including this. **??** presents background and the state of the art for WebAssembly and Artificial Software Diversification. **??** describes our technical contributions, faced challenges and engineering decisions carried out to implement our artifacts. **??** describes the methodology followed to answer the three main research questions driving this thesis. **??** details the main results of this work. **??** concludes and discuss future work. In addition, this dissertation contains the collection of research papers previously mentioned in this chapter.

02 BACKGROUND & STATE OF THE ART

This chapter discusses the state of the art in the areas of *WebAssembly* and *Software Diversification*. In **??** we discuss the WebAssembly language, its motivation, how WebAssembly binaries are generated, the language specification, and security-related issues. In **??**, we present a summary of Software Diversification, its foundational concepts and highlighted related works. We select the discussed works by their novelty, critical insights, and representativeness of their techniques. In **??**, we finalize the chapter by highlighting open challenges in state-of-the-art related works.

2.1 WebAssembly overview

JavaScript is currently used in all modern web browsers to allow client-side scripting. However, due to the complexity of this language and to gain in performance and its security flaws, several alternatives appeared. For example, Java applets were introduced on web pages late in the 90's to execute Java bytecode in the client side [**?**]. Similarly, Microsoft made two attempts with ActiveX in 1996 [**?**], and with Silverlight in 2007 [**?**]. All these attempts failed to persist or had low adoption, mainly due to security issues and the lack of consensus on the community of browser vendors.

In 2014, Alon Zakai and colleagues proposed the Emscripten tool [**?**]. Emscripten used a strict subset of JavaScript, asm.js, to allow low level code such as C to be compiled to JavaScript. Asm.js was first announced as an LLVM backend [**?**]. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [**?**] compared to standard JavaScript code. The main reason why asm.js is faster, is that it limits the language features to those that can be optimized in the LLVM pipeline. Besides, it removes the majority of the dynamic characteristics of the language, limiting it to numerical types, top-level functions, and one large array in the memory directly accessed as raw data. Since asm.js is a subset of JavaScript it was compatible with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standarization. The work of Van Es et al. [**?**] proposed to shrink JavaScript to asm.js in a source-to-source strategy, closing the cycle and extending the fact that asm.js was mainly a compilation target for C/C++ code. Moreover, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the

parsing and the recompilation of the JavaScript code which implies a significant overhead. Consequently, the asm.js initiative, the W3C publicly announced the WebAssembly (Wasm) language in 2015. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially consolidated later by the work of Haas et al. [**?**] in 2017. The announcement of WebAssembly marked the first step into the standarization of bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it outperforms asm.js [**?**]. Since 2017, the adoption of WebAssembly keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm Wasmlike is the case of a full Minecraft version [2]; and the case of Blazor [3], a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

■ 2.1.1 From source to Wasm

All WebAssembly programs are compiled ahead-of-time from source languages. LLVM includes Wasm as a backend since release 7.1.0 published in May 2019[4], supporting a broad range of frontend languages such as C/C++, Rust, Go or AssemblyScript[5]. The resulting binary works similarly to a traditional shared library, it includes instruction codes, symbols and exported functions. In **??**, we illustrate the workflow from the creation of Wasm binaries to their execution in the browser. The process starts by compiling the source code program to Wasm (Step ①). This step includes ahead-of-time optimizations such as optimizations in the LLVM toolchain.

The step ② builds the standard library for Wasm usually as JavaScript code. This code includes the external functions that the Wasm binary needs for its execution inside the host engine. For example, the functions to interact with the DOM of the HTML page are imported in the Wasm binary during its call from the JavaScript code. The standard library can be manually written, however, compilers like Emscripten, Rust and Binaryen can generate it automatically, making this process completely transparent to developers.

Finally, the third step (Step ③), includes the compilation and execution of the client-side code. Most of the browser engines compile either the Wasm and JavaScript codes to machine code. In the case of JavaScript, this process involves JIT and hot code replacement during runtime. For Wasm, since it is closer to machine code, and it is already optimized, this process is a one-to-one mapping. For instance, in the case of V8, the compilation process only applies simple and fast optimizations such as constant folding and dead code removal. Once V8 completes

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2] `https://satoshinm.github.io/NetCraft/`

[3] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

[4] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`

[5] subset of the TypeScript language

the compilation process, the generated machine code for Wasm does not change anymore and is the same used along all its executions. This analysis was validated by conversations with the V8's dev team and by experimental studies in our previous contributions [?].

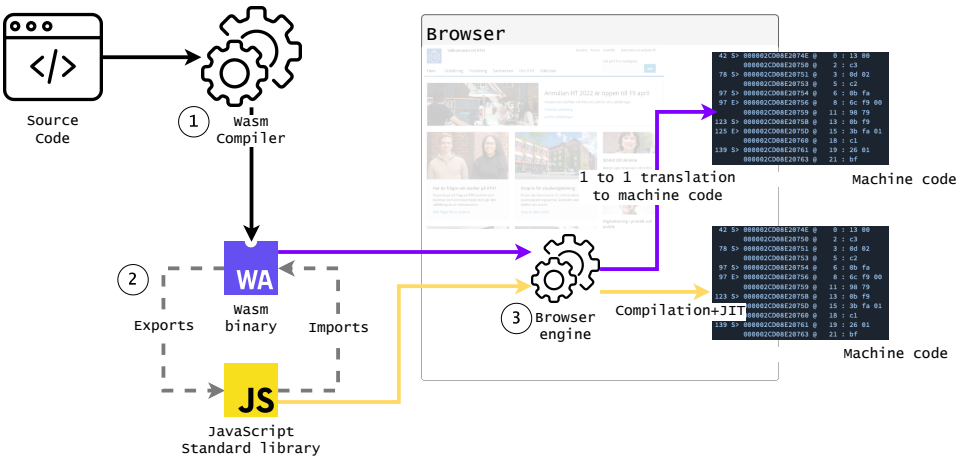


Figure 2.1: WebAssemblybuilding, compilation in the host engine and execution.

Wasm can execute directly and is platform independent. As such it is useful for IoT and Edge computing [? ?]. For instance, Cloudflare and Fastly adapted their platforms to provide Function as a Service (FaaS) directly with WebAssembly. In this case, the standard library, instead of JavaScript, is provided by any other language stack that the host environment supports. In 2019, the Bytecode Alliance [6] proposed WebAssembly System Interface (WASI) [?]. WASI is the foundation to build Wasm code outside the browser with a POSIX system interface platform. WASI standarizes the adoption of WebAssembly in heterogeneous platforms [?].

■ 2.1.2 WebAssembly specification

WebAssembly defines its own Instruction Set Architecture (ISA) [?]. It is an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types[7], each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

In ?? and ?? we illustrate a C program and its compilation to Wasm. The C function contains: heap allocation, external function declaration and the definition

[6] https://bytecodealliance.org/
[7] https://webassembly.github.io/spec/core/binary/modules.html#sections

of a function with a loop, conditional branching, function calls and memory accesses. The code in **??** is in the textual format for the generated Wasm. The module in this case first defines the signature of the functions(**??**, **??** and **??**) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (**??**, **??** and **??**). The definition of the function (**??**) and its body follows the last import declaration at **??**.

The function body is composed by local variable declarations and typed instructions that are evaluated in a virtual stack (Line 7 to Line 32 in **??**). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of **??**, the result value of the main function is the calculation of the last instruction, `i32.add` at **??**. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. As the listing shows, instructions are annotated with a numeric type.

Wasm manages the memory in a restricted way. A Wasm module has a linear memory component that is accessed as `i32` pointers and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in **??**. The memory access is illustrated in **??**. This memory is usually bound in browser engines to 2Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in **??** and **??**). Therefore, this ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (**??**) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (**??**) or using a static table of function declarations. This latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, for which the Wasm's compiler is reponsible of populating the static table of functions.

In Wasm, all instructions are grouped into blocks, being the start of a function the root block. Two consecutive block declarations can be appreciated in **??** and **??** of **??**. Control flow structures jump between block boundaries and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (**??** and **??**). By design, each block executes independently and cannot execute or refer to outer block values. This is guarantieed by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (**??** and **??**) and table break. Each break instruction can only jump to one of its enclosing blocks. For example, in **??**, **??** forces the execution to jump to the end of the first

Listing 2.1: Example C function.

```c
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.2: WebAssembly code for **??**.

```
(module
  (type (;0;) (func (param f32) (result i32)))
  (type (;1;) (func))
  (type (;2;) (func (result i32)))
  (import "env" "ftoi" (func $ftoi (type 0)))
  (func $main (type 2) (result i32)
    (local i32 i32)
    i32.const -1000
    local.set 0
    block  ;label = @1;
      loop  ;label = @2;
        i32.const 0
        local.get 0
        i32.add
        i32.load
        local.tee 1
        i32.const 101
        i32.ge_s
        br_if 1 ;@1;
        local.get 0
        i32.const 4
        i32.add
        local.tee 0
        br_if 0 ;@2;
      end
      i32.const 0
      return
    end
    f32.const 0x1.9147aep+3
    call $ftoi
    local.get 1
    i32.add)
  (memory (;0;) 1)
  (global (;4;) i32 (i32.const 1000))
  (export "memory" (memory 0))
  (export "A" (global 2))
  (data $data (0) "\00\00\00\00...")
)
```

block at **??** if the value at the top of the stack is greater than zero.

■ 2.1.3 WebAssembly security

As we described, WebAssembly is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [**?**] and makes it easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

WebAssembly is vulnerable [**?**]. Implementations in both browsers and standalone runtimes [**?**] are vulnerable. Genkin et al. demonstrated that Wasm

could be used to exfiltrate data using cache timing-side channels [**?**]. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [**?**] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap using stackoverflow. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for WebAssembly. Recently, Stiévenart and colleagues demonstrate that C/C++ source code vulnerabilities can be ported to Wasm [**?**]. Several proposals for extending WebAssembly in the current roadmap could address some existing vulnerabilities. For example, having multiple memories [8] could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors [9].

■ 2.2 Software Diversification

Software Diversification has been widely studied in the past decades. This section discusses its state of the art. Software diversification consists in synthesizing, reusing, distributing, and executing different, functionally equivalent programs. According to the survey by Baudry et al. [**?**], the motivation for software diversification can be separated in five categories: reusability [**?**], software testing [**?**], performance [**?**], fault tolerance [**?**] and security [**?**]. Our work contributes to the latter two categories. In this section we discuss related works by highlighting how they generate diversification and how they put it into practice.

There are two primary sources of software diversification: Natural Diversity and Artificial Diversity[**?**]. This work contributes to the state of the art of Artificial Diversity, which consists of synthesizing software. This thesis is founded on the work of Cohen in 1993 [**?**] as follows.

■ 2.2.1 Variants' generation

Cohen et al. proposed to generate artificial software diversification through mutation strategies. A mutation strategy is a set of rules to define how a specific component of software development should be changed to provide a different yet functionally equivalent program. Cohen et al. proposed 10 concrete transformation strategies that can be applied at fine-grained levels. All described strategies can be mixed together. They can be applied in any sequence and recursively, providing a richer diversity environment. We summarize them, complemented with the work of Baudry et al. [**?**] and the work of Jackson et al. [**?**], in 5 strategies.

(S1) *Equivalent instructions replacement* Semantically equivalent code can replace pieces of programs. This strategy replaces the original code with equivalent

[8] `https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/Overview.md`

[9] `https://webassembly.org/roadmap/`

arithmetic expressions or injects instructions that do not affect the computation result. There are two main approaches for generating equivalent code: rewriting rules and exhaustive searching. The replacement strategies are written by hand as rewriting rules for the first one. A rewriting rule is a tuple composed of a piece of code and a semantic equivalent replacement. For example, Cleemput et al. [**?**] and Homescu et al. [**?**] insert NOP instructions to generate statically different variants. In their works, the rewriting rule is defined as `instr => (nop instr)`, meaning that `nop` operation followed by the instruction is a valid replacement . On the other hand, exhaustive searching samples all possible programs for a specific language. In this topic, Jacob et al. [**?**] proposed the technique called superdiversification for x86 binaries. The superdiversification strategy proposed by Jacob and colleagues performs an exhaustive search of all programs that can be constructed from a specific language grammar. If one of the generated programs is equivalent to the original program, then it is reported as a variant. Similarly, Tsoupidi et al. [**?**] introduced Diversity by Construction, a constraint-based compiler to generate software diversity for MIPS32 architecture.

(S2) *Instruction reordering* This strategy reorders instructions or entire program blocks if they are independent. The location of variable declarations might change as well if compilers re-order them in the symbol tables. It prevents static examination and analysis of parameters and alters memory locations. In this field, Bhatkar et al. [**?** **?**] proposed the random permutation of the order of variables and routines for ELF binaries.

(S3) *Adding, changing, removing jumps and calls* This strategy creates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [**?**] mainly illustrated the case by inserting bogus jumps in programs. Pettis and Hansen [**?**] proposed to split basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [**?**] de-inline basic blocks of code as an LLVM pass. In their approach, each de-inlined code is transformed into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [**?**] extended their previous approach [**?**], replacing function calls by indirect pointer calls in C source code, allowing post binary reordering of function calls. Recently, Romano et al. [**?**] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm function.

(S4) *Program memory and stack randomization* This strategy changes the layout of programs in the host memory. Also, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [**?** **?**] propose to randomize the base addresses of applications and the library memory regions in ELF binaries. Tadesse Aga and Autin [**?**] and Lee et al. [**?**] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [**?**] propose to separate a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [**?**]

transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization.

(S5) *ISA randomization and simulation* This strategy uses a key to cypher the original program binary into another encoded binary. Once encoded, the program can be decoded only once at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [**?**] and Barrantes et al. [**?**] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [**?**] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [**?**] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Code obfuscation [**?**] can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as it is in any client. Yet, both strategies are meant to tackle program analysis from potential attackers.

■ 2.2.2 Variants' equivalence

Equivalence checking between program variants is an essential component for any program transformation task, from checking compiler optimizations [**?**] to the artificial synthesis of programs discussed in this chapter. Equivalence checking proves that two pieces of code or programs are semantically equivalent [**?**]. Cohen [**?**] simplifies this checking by enunciating the following property: two programs are equivalent if given identical input, they produce the identical output. We use this same enunciation as the definition of *functional equivalence* along with this dissertation. Equivalence checking in Software Diversification aims to preserve the original functionality for programs while changing observable behaviors. For example, two programs can be statically different or have different execution times and provide the same computation.

The equivalence property is often guaranteed by construction. For example, in the case illustrated in **??** for Cleemput et al. [**?**] and Homescu et al. [**?**], their transformation strategies are designed to generate semantically equivalent program variants. However, this process is prone to developer errors, and further validation is needed. For example, the test suite of the original program can be used to check the variant. If the test suite passes for the program variant [**?**], can be considered equivalent to the original. However, it is limited due to the need for a preexisting test suite. When the test suite does not exist, another technique is needed to check for equivalence.

If there is no test suite or the technique does not inherently implement the equivalence property, the previously mentioned works use theorem solvers (SMT

solvers) [**?**] to prove equivalence. For SMT solvers, the main idea is to turn the two code variants into mathematical formulas. The SMT solver checks for counter-examples. When the SMT solver finds a counter-example, there exists an input for which the two mathematical formulas return a different output. The main limitation of this technique is that all algorithms cannot be translated to a mathematical formula, for example, loops. Yet, this technique tends to be the most used for no-branching-programs checking like basic block and peephole replacements [**?**].

Another approach to check equivalence between two programs similar to using SMT solvers is by using fuzzers [**?**]. Fuzzers randomly generate inputs that provide different observable behavior. If two inputs provide a different output in the variant, the variant and the original program are not equivalent. The main limitation for fuzzers is that the process is remarkably time-expensive and requires the introduction of oracles by hand.

■ 2.2.3 Usages of Software Diversity

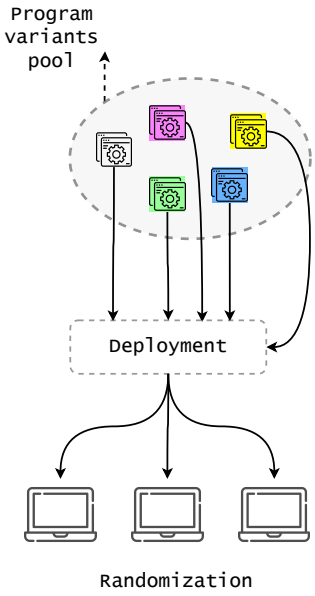
After program variants are generated, they can be used in two main scenarios: Randomization or Multivariant Execution(MVE) [**?**]. In **??** and **??** we illustrate both scenarios.

(U1) *Randomization:* In the context of our work *Randomization* refers to the ability of a program to be served as different variants to different clients. In the scenario of **??**, a program is selected from the collection of variants (program's variant pool), and at each deployment, it is assigned to a random client. Jackson et al. [**?**] compare the variant's pool in Randomization with a herd immunity, since vulnerable binaries can affect only part of the client's community.

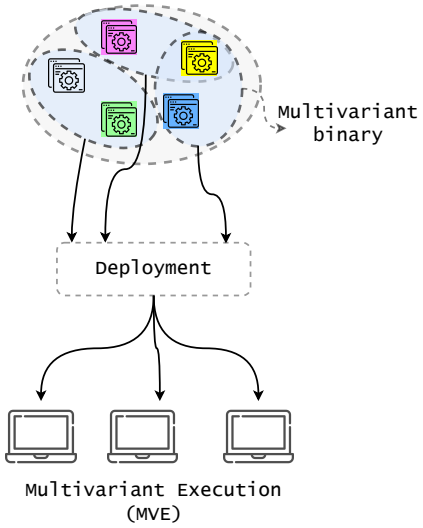
El-Khalil and colleagues [**?**] propose to use a custom compiler to generate different binaries out of the compilation process. El-Khalil and colleagues modify a version of GCC 4.1 to separate a conventional stack into several component parts, called multistacks. On the same topic, Aga and colleagues [**?**] propose to generate program variants by randomizing its data layout in memory. Their approach makes each variant to operate the same data in memory with different memory offsets. Remarkably, the Polyverse company [10] materialize randomization at the commercial level in real life. They deliver a unique Linux distribution compilation for each of its clients by scrambling the Linux packages at the source code level.

Virtual machines and operating systems can be also randomized. On this topic, Kc et al. [**?**] create a unique mapping between artificial CPU instructions and real ones. Their approach makes possible the assignation of different variants to specific target clients. Similarly, Xu et al. [**?**] recompile the Linux Kernel to reduce the exposure time of persistent memory objects, increasing the frequency of address randomization.

[10] `https://polyverse.com/`



(a) Randomization scenario. Given a pool of program variants, one variant is deployed per host. Each deployment randomly selects which variant is assigned to each host. The same program variant is executed in the host at every program invocation between deployments.



(b) Multivariant Execution scenario. Given a pool of program variants, a sample of the pool is packaged in a multivariant binary that is deployed. Each deployment randomly selects which multivariant binary is assigned to each host. A variant from the multivariant binary is randomly executed at runtime in the host, .

Figure 2.2: Software Diversification usages.

(U2) *Multivariant Execution (MVE):* Multiple program variants are composed in one single binary (multivariant binary) [**?**]. Each multivariant binary is randomly deployed to a client. Once in the client, the multivariant binary executes its embedded program variants at runtime. **??** illustrates this scenario.

The execution of the embedded variants can be either in parallel to check for inconsistencies or a single program to randomize execution paths [**?**]. Bruschi et al. [**?**] extended the idea of executing two variants in parallel with not-overlapping and randomized memory layouts. Simultaneously, Salamat et al. [**?**] modified a standard library that generates 32-bit Intel variants where the stack grows in the opposite direction, checking for memory inconsistencies. Notably, Davi et al. proposed Isomeron [**?**], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is

running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. The previously mentioned works showed the benefits of exploiting the limit case of executing only two variants in a multivariant environment. Agosta et al. [**?**] and Crane et al. [**?**] used more than two generated programs in the multivariant composition, randomizing software control flow at runtime.

Both scenarios have demonstrated to harden security by tackling know vulnerabilities such as (JIT)ROP attacks [**?**] and power side-channels [**?**]. Moreover, Artificial Software Diversification is a preemptive technique for yet unknown vulnerabilities [**?**]. Our work contributes to both usages scenarios for WebAssembly.

■ 2.3 Open challenges

In **??** we list the related work on Artificial Software Diversification discussed along with this chapter. The first and second columns in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (**??**, **??**, **??**, **??**, **??**, **??** and **??**). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. In the following we enumerate the open challenges we have found in the literature research:

1. *WebAssembly is vulnerable:* WebAssembly is a novel technology, yet vulnerable. The adoption of defenses for it is still under development [**?**] and has a low pace.

2. *Software monoculture phenomenon:* WebAssembly is evolving further web browsers as a technology to support Edge-Cloud computing platforms. The Edge-Cloud computing model is based on replicating the same binary along with all computing nodes in a worldwide scale. Therefore, potential vulnerabilities are spread, highlighting a clear monoculture phenomenon [**?**].

3. *Lack of Software Diversification for WebAssembly*: Software Diversification has demonstrated to provide protection for known and yet-unknown vulnerabilities. Yet, only one software diversity approach has been applied to the context of WebAssembly [**?**]. Moreover, as we illustrate in **??**, the existing works for Software Diversification do not contribute directly to WebAssembly.

4. *Lack of research on MVE for WebAssembly:* WebAssembly has a growing adoption for Edge platforms. However, researching on MVE in a distributed setting like the Edge has been less researched. Only Voulimeneas et al. [**?**] recently proposed a multivariant execution system by parallelizing the

execution of the variants in different machines for the sake of efficiency. This work

5. *Porting current contributions:* The preexisting works based on the LLVM pipeline cannot be extended to Wasm due to technical challenges. The main reason is that previous works contribute to LLVM versions released before the inclusion of Wasm as an LLVM architecture. On the other hand, there is no mentioned work merging all mutation strategies in one solution.

■ Conclusions

In this chapter, we presented the background on the WebAssembly language, including its security issues and related work. This chapter aims to settle down the foundation to study automatic diversification for WebAssembly. We highlighted related work on Artificial Software Diversification, showing that it has been widely researched, not being the case for WebAssembly. On the other hand, current available implementations for Software Diversification cannot be directly ported to Wasm. The current limitations on security and the lack of software diversity approaches for WebAssembly motivate our work. We place our contributions in the field of artificial diversity. In **??** we describe the technical details that lead our contributions. Besides, the impact of our contributions is evaluated by following the methodology described in **??**.

Authors	??	??	??	??	??	??	??	Main technical contribution
Pettis and Hansen [?]		✓		✓		✓		Custom Pascal compiler for PA-RISC architecture
Chew and Song [?]			✓			✓		Linux Kernel recompilation.
Kc et al. [?]					✓			Linux Kernel recompilation.
Barrantes et al. [?]					✓	✓		x86 to x86 transformations using Valgrind
Bhatkar et al. [?]	✓	✓		✓		✓		ELF binary transformations
El-Khalil and Keromytis [?]						✓		custom GCC compiler for x86 architecture
Bhatkar et al. [?]	✓	✓		✓		✓		C/C++ source to source transformations and ELF binary transformations
Younan et al. [?]				✓				custom GCC compiler
Bruschi et al. [?]				✓		✓		ELF binary transformations.
Salamat et al. [?]			✓				✓	Custom GNU compiler
Jacob et al. [?]	✓	✓						x86 to x86 transformations
Salamat et al. [?]				✓			✓	x86 to x86 transformations
Amarilli et al. [?]	✓				✓	✓		Polymorphic code generator for ARM architecture
Jackson [?]	✓					✓	✓	LLVM compiler, only backend for x86 architecture
Cleemput et al. [?]	✓					✓		x86 to x86 transformations
Homescu et al. [?]	✓					✓		LLVM 3.1.0†
Crane et al. [?]	✓	✓	✓				✓	LLVM, only backend for x86 architecture
Davi et al. [?]							✓	Windows DLL instrumentation
Couroussé et al. [?]	✓	✓			✓	✓		Custom GCC compiler targeting microcontrollers
Lu et al. [?]				✓			✓	GNU assembler for Linux kernel
Belleville et al. [?]	✓			✓		✓		Only C language frontend, LLVM 3.8.0†
Aga et al. [?]				✓		✓		Data layout randomization, LLVM 3.9†
Österlund et al. [?]				✓			✓	Linux Kernel recompilation.
Xu et al. [?]				✓		✓		Custom kernel module in Linux OS
Lee et al. [?]				✓		✓		LLVM 12.0.0 backend for x86
Romano et al. [?]			✓			✓		JavaScript and Wasm intermixing

† Notice that LLVM only supports WebAssembly backend from release 7.1.0

Table 2.1: The first and second columns in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (??, ??, ??, ??, ??, ?? and ??). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order.

03 AUTOMATIC DIVERSITY FOR WEBASSEMBLY

We aim to create artificial software diversity for WebAssembly, by providing tools to make the process easier and feasible for developers and researchers. According to our exhaustive literature review, no software provides artificial software diversification for WebAssembly. Therefore, we need to enunciate the engineering foundation to implement the strategies defined in **??**. Our implementations are part of the contributions of this thesis. We provide two tools that complement this work: CROW and MEWE. First, the former tool generates WebAssembly program variants statically at compile time to provide randomization. The latter tool provides the tooling to generate MVE binaries for WebAssembly. In this chapter, we describe our technical contributions. In **??** we enunciate how the current state-of-the-art lead us to contribute with Software Diversification through LLVM. We follow by describing our two contributions and their main technical insights in **??** and **??**. Besides, we describe a new transformation strategy as part of our contributions.

■ 3.1 Global approach

The work of Hilbig et al. [**?**] at 2021 influences our design decisions. According to their work, 70% of the WebAssembly binaries in the wild are created with LLVM-based compilers. Therefore, we provide artificial software diversity for WebAssembly through LLVM. Other solutions would have been to diversify at the source code level or the WebAssembly binary level. However, the former would limit the applicability of our work. We propose the latter for future works.

LLVM is a compound of three main components [**?**]. First, the frontend (compilers such as clang and rustc) converts the program source code to LLVM intermediate representation (LLVM IR). Second, optimization and transformation processes improve the LLVM IR. Third and final, the backend component is in charge of generating the target machine code. In **??** we show how we use the LLVM pipeline in our contributions, which are highlighted as dashed squares.

The global workflow in **??** starts by receiving the source code. Then the LLVM frontend transforms it into LLVM IR representation ①. We alter the LLVM pipeline that compiles source code to Wasm by introducing a diversifier component.

The diversifier generates LLVM IR variants from the output of the frontend ②. The LLVM IR variants are inputs for our customized Wasm backend. The diversifier and the custom Wasm LLVM backend compose CROW, which creates

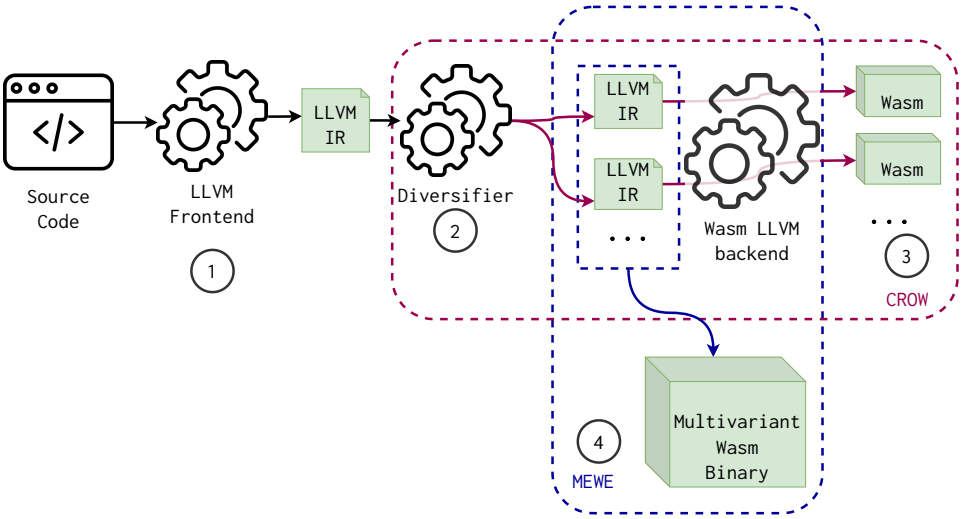


Figure 3.1: Generic workflow to create WebAssembly program variants.

WebAssembly program variants out of a source code program ③. In addition, an orthogonal tool comes from the generation of LLVM IR variants at Step ②. MEWE [**?**], merges and creates multivariant binaries to provide MVE for WebAssembly ④.

■ 3.2 CROW: Code Randomization of WebAssembly bytecode

This section describes the red squared tooling in **??** named, CROW [**?**]. CROW is a tool tailored to create semantically equivalent WebAssembly variants from an LLVM front-end output. Using a custom Wasm LLVM backend, it generates the Wasm binary variants.

In **??**, we describe the workflow of CROW to create program variants. The Diversifier in CROW is composed by two main processes, *exploration* and *combining*. The *exploration* process operates at the instruction level for each function in its input LLVM. For each instruction, CROW produces a collection of functionally equivalent code replacements. In the *combining* stage, CROW assembles the code replacements to generate different LLVM IR variants. CROW generates the LLVM IR variants by traversing the power set of all possible combinations of code replacements. Finally, the custom Wasm LLVM backend compiles the assembled LLVM IR variants into WebAssembly binaries. In the following, we describe our design decisions. All our implementation choices

are based on one premise: *each design decision should increase the number of WebAssembly variants that CROW creates.*

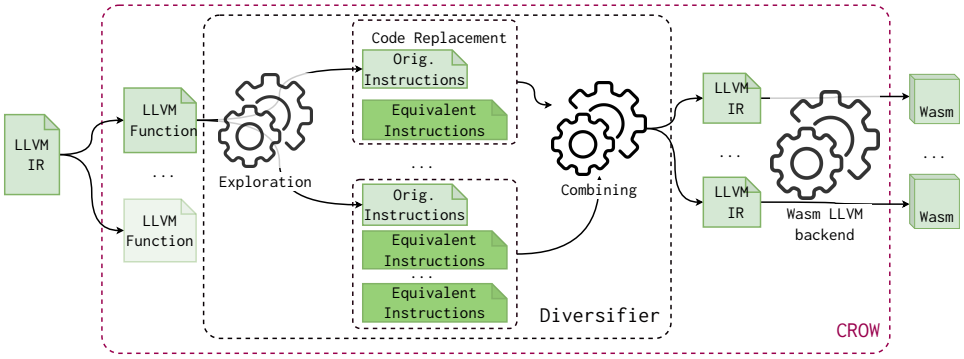


Figure 3.2: CROW components following the diagram in **??**. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them.

■ 3.2.1 Exploration

The primary component of CROW's exploration process is its code replacements generation strategy. The diversifier implemented in CROW is based on the superdiversifier of Jacob et al. [**?**]. A superoptimizer focuses on *searching* for a new program that is faster or smaller than the original code while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [**?**] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by code size in ascending order. If any of these programs are found to perform the same function as the source program, the search halts. On the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We use the superdiversifier idea of Jacob and colleagues to implement CROW because two main reasons. First, the code replacements generated by this technique outperform diversification strategies based on hand-written rules. Besides, this technique is fully automatic. Second, there is a battle-tested superoptimizer for LLVM, Souper [**?**]. This latter makes feasible the construction of a generic LLVM superdiversifier.

We modify Souper to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem

solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation the optimal replacement result of the searching. We keep more equivalent replacements during the searching by removing the halting criteria. Instead, we limit the searching for a replacement with timeout and the replacement's size. Our customized Souper reports a new code replacement as soon as an equivalent transformation is found.

Notice that the searching space exponentially increases with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions with no correspondence in the WebAssembly backend. This decision reduces the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more variants.

3.2.2 Constant inferring

One code transformation strategy of Souper does *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on boolean-valued variables that are used to control branches. By extending Souper as a superdiversifier, we add this transformation strategy as a new mutation strategy to the ones defined in **??**.

After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification. Let us illustrate the case with an example. The Babbage problem code in **??** is composed of a loop that stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 3.1: Babbage problem.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) % 1000000
            ↪ != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n", current);
8       return 0 ;
9   }
```

Listing 3.2: Constant inferring transformation over the original Babbage problem in **??**.

```
int babbage() {
    int current = 25264;

    printf ("The number is %d\n", current);
    return 0 ;
}
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large. The original Souper deals with this case, generating the program in **??**. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final

compilation. The new program in **??** is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime. [1]

■ 3.2.3 Removing latter optimizations for LLVM

During the implementation of CROW, we have the premise of removing all built-in optimizations in the LLVM backend that could reverse Wasm variants. Therefore, we modify the WebAssembly backend in addition to the extended Souper. We disable all optimizations in the WebAssembly backend that could reverse the superoptimizer transformations, such as constant folding and instructions normalization.

■ 3.3 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [**?**]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE. The tool generates application-level multivariant binaries without changing the operating system or WebAssembly runtime. MEWE creates an MVE by intermixing functions for which CROW generates variants, as step ② in **??** shows. CROW generates each one of these variants with fine-grained diversification at the instruction level, applying the majority of the strategies discussed in **??** and *constant inferring*. MEWE adds a new mutation strategy. It inlines function variants when appropriate, resulting in call stack diversification at runtime.

In **??** we zoom MEWE from the blue highlighted square in **??**. MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE merges the LLVM IR variants created by CROW and creates an LLVM multivariant binary. The merging of the variants intermixes the calling of function variants, making possible the execution path randomization.

The Mixer augments the LLVM multivariant binary with a random generator. The random generator is needed to perform the execution-path randomization. Also, *The Mixer* fixes the entrypoint in the multivariant binary. Finally, MEWE generates a standalone multivariant WebAssembly binary using the same custom Wasm LLVM backend from CROW. Once generated, the generated multivariant WebAssembly binary can be deployed to any WebAssembly engine.

■ 3.3.1 Multivariant generation

The key component of MEWE consists in combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The

[1] Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR. Also, notice that the two programs in the example follow the definition of *functional equivalence* discussed in **??**.

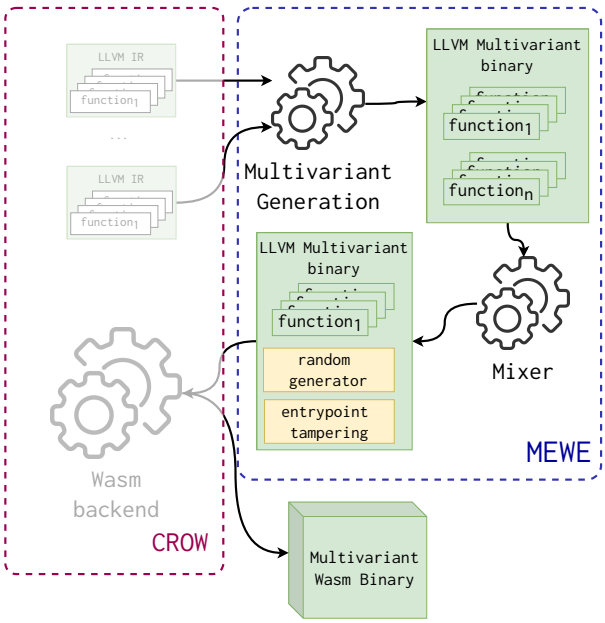


Figure 3.3: Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a WebAssembly multivariant binary ready to be deployed.

core idea is introducing one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 1. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph* $\langle N, E \rangle$ *where the nodes in* N *represent all the functions in the binary and an edge* $(f_1, f_2) \in E$ *represents a possible invocation of* f_2 *by* f_1 *[**?**], where the nodes are typed. The nodes in* N *have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

In **??**, we show the original static call graph for and original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The grey nodes represent function variants, the green nodes function

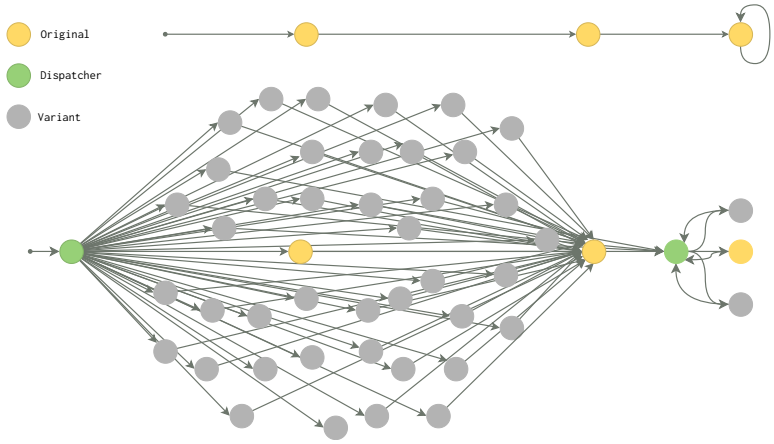


Figure 3.4: Example of two static call graphs. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in grey), dispatchers (in green), and original functions (in yellow).

dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

In **??**, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of **??**. It first calls the random generator, which returns a value used to invoke a specific function variant. We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution-based attacks [**?**]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [**?**]. This also allows MEWE to inline function variants inside the dispatcher instead of defining them again. Here we trade security over performance since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

■ 3.3.2 The Mixer

MEWE has four specific objectives: link the LLVM multivariant binary, inject a random generator, tamper the application's entrypoint, and merge all these components into a multivariant WebAssembly binary. We use the Rustc compiler[2]

[2] `https://doc.rust-lang.org/rustc/what-is-rustc.html`

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.3: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in **??**.

to orchestrate the mixing. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. The Mixer creates a new entrypoint for the binary called *entrypoint tampering*. It wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

■ 3.4 Accompanying Source Code

This thesis is accompanied by the source code of both contributions, CROW and MEWE. The source code is accessible through the links:

1. CROW: `https://github.com/KTH/slumps`
2. MEWE: `https://github.com/Jacarte/MEWE`

Our software artifacts are licensed under the MIT License. The dependent source codes, such as LLVM, are licensed under their original licenses.

■ Conclusions

This chapter discusses the technical details of the tools implemented for our main contributions. We describe how CROW generates program variants for the sake of software diversification. We propose a global architecture for a generic LLVM superdiversifier We introduce a new mutation strategy that is a consequence of

retargeting Souper as a superdiversifier. Besides, we dissect MEWE and how it creates an MVE system. In **??** we discuss the methodology we follow to evaluate how CROW and MEWE create software diversification.

04 METHODOLOGY

In this chapter, we present our methodology to answer the research questions enunciated in **??**. We investigate three research questions. In the first question, we aim to investigate the static differences between variants. We evaluate the code properties the lead less or more software diversification. Our second research question focuses on comparing their behavior during their execution, complementing our first research question. The generated variants should be statically different, but also should provide different observable behavior. The final research question evaluates the feasibility of using the program variants in security-sensitive environments. We evaluate our generated program variants in an Edge-Cloud computing platform proposing a novel multivariant execution approach.

The main objective of this thesis is to study the feasibility of automatically creating program variants out of preexisting program sources. To achieve this objective, we use the empirical method [**?**], proposing a solution and evaluating it through quantitative analyzes in case studies. We follow an iterative and incremental approach on the selection of programs for our corpora. To build our corpora, we find a representative and diverse set of programs to generalize, even when it is unrealistic following an empirical approach, as much as possible our results. We first enunciate the corpora we share along this work to answer our research questions. Then, we establish the metrics for each research question, set the configuration for the experiments, and describe the protocol.

4.1 Corpora

Our experiments assess the impact of artificially created diversity. The first step is to build a suitable corpus of programs' seeds to generate the variants. Then, we answer all our research questions with three corpora which follow two main properties: 1) *functionally diverse:* the selection of the programs is not biased by functionally fixed tasks, for example, the programs in one of our corpora solve from the *Babbage* problem to *Convex Hull* calculation; and 2) *representative:* our corpora have 3021 programs that can be ported to WebAssembly, representing approximately 40% of the unique Wasm binaries in the wild [**?**].

We build our three corpora in an escalating strategy based on the merging of our previous publications. The first corpus is diverse and contains simple programs in terms of code size, making them easy to manually analyze. The second corpus is a project meant for security-sensitive applications. The third corpus is a QR encoding

decoding algorithm. In the following, we describe the filtering and description of each corpus.

1. **Rosetta**: We take programs from the Rosetta Code project[1]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains many tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from the Rosetta Code, representing 989 programs as of 01/26/2020. We then apply several filters: the programs should successfully compile and, they should not require user inputs to automatically execute them, the programs should terminate and should not result in non-deterministic results.

   The result of the filtering is a corpus of 303 C programs. All programs include a single function in terms of source code. These programs range from 7 to 150 lines of code.

2. **Libsodium**: This project is encryption, decryption, signature, and password hashing library implemented in 102 separated modules. The modules have between 8 and 2703 lines of code per function. This project is selected based on two main criteria: first, its importance for security-related applications, and second, its suitability to collect the modules in LLVM intermediate representation.

3. **QrCode**: This project is a QrCode and MicroQrCode generator written in Rust. This project contains 2 modules having between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for collecting the modules in their LLVM representation. Remarkably, this project increases the complexity of the previously selected projects due to its integration with the generation of images.

In **??** we listed the corpus name, the language of the programs in the corpus, the number of modules, the total number of functions, the range of lines of code, and the original location of the corpus.

■ 4.2 RQ_1. To what extent can we artifically generate program variants for WebAssembly?

This research question investigates whether we can artificially generate program variants for WebAssembly. We use CROW to generate variants from an original program, written in C/C++ in the case of Rosetta corpus and LLVM bitcode modules in the case of Libsodium and QrCode. In **??** we illustrate the workflow to generate WebAssembly program variants. We pass each function of the corpora to

[1] `http://www.rosettacode.org/wiki/Rosetta_Code`

Corpus	Lang.	No. modules	No. functions	LOC range	Location
Rosetta	C	-	303	7 - 150	`https://github.com/KTH/slumps/tree/master/benchmark_programs/rossetta/valid/no_input`
Libsodium	LLVM IR + Rust	102	869	8 - 2703	`https://github.com/jedisct1/libsodium/tree/2b5f8f2b6810121c2d9a8cc8a392e01f4d3de433`
QrCode	LLVM IR + Rust	2	1849	4 - 725	`https://github.com/kennytm/qrcode-rust/commit/faa4397ba7c5f441cb9a2b436c1e84a0d52ae942`
Total			3021		

Table 4.1: Corpora description. The table is composed by the name of the corpus, programming language of the programs in the corpus, the number of modules, the number of functions, the lines of code range and the location of the corpus.

CROW as a program to diversify. To answer RQ1, we study the outcome of this pipeline, the generated WebAssembly variants.

Metrics

To assess our approach's ability to generate WebAssembly binaries that are statically different, we compute the number of variants and the number of unique variants for each original function of each corpus. On top, we define the aggregation of these former two values to quantitatively evaluate RQ1 at the corpus level.

We start by defining what a program's population is. This definition can be applied in general to any collection of variants of the same program. All definitions are based upon bytecodes and not the source code of the programs.

Definition 2. *Program's population $M(P)$:* Given a program P and its generated variants v_i, the program's population is defined as.

$$M(P) = \{v_i \text{ where } v_i \text{ is a variant of P}\}$$

Notice that, the program's population includes the original program P.

Beyond the program's population, we also want to compare how many program variants are unique. The subset of unique programs in the program's population hints how the variants are different between them and not only against the original

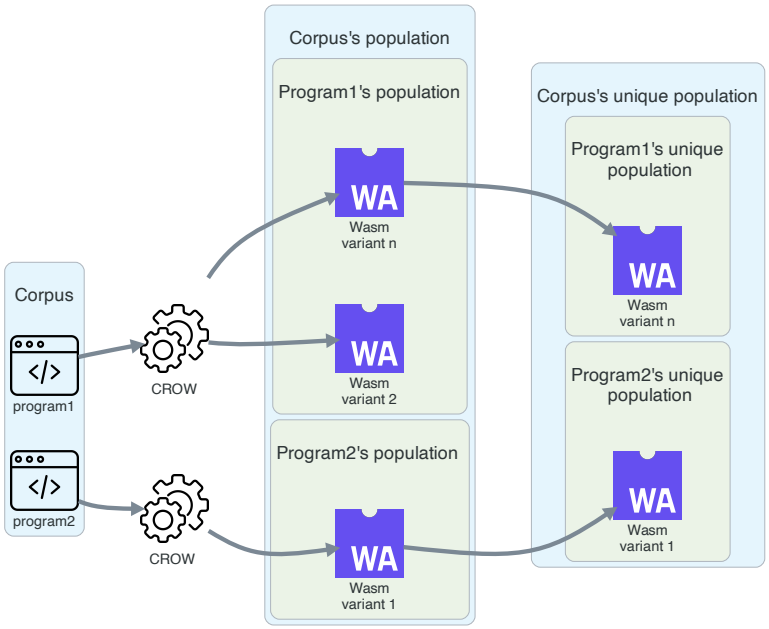


Figure 4.1: The program variants generation for RQ1.

program. For example, imagine a program P with two program variants V_1 and V_2, the program population is composed by $\{P, V_1$ and $V_2\}$, where V_1 is different from P, and V_2 is different from P. Either, if V_1 is equal or different from V_2, the program's population still be the same.

Definition 3. *Program's unique population $U(P)$:* Given a program P and its program's population $M(P)$, the program's unique population is defined as.

$$U(P) = \{v \ \in \ M(P)\}$$

such that $\forall v_i, v_j \in U(P)$, $md5sum(v_i) \neq md5sum(v_j)$. $Md5sum(v)$ is the md5 hash calculated over the bytecode stream of the program file v. Notice that, the original program P is included in $U(P)$.

Metric 1. *Program's population size $S(P)$:* Given a program P and its program's population $M(P)$ according to **??**, the program's population size is defined as.

$$S(P) = |M(P)|$$

Metric 2. *Program's unique population size $US(P)$:* Given a program P and its program's unique population $U(P)$ according to **??**, the program's unique population size is defined as.

$$US(P) = |U(P)|$$

Metric 3. *Corpus population size $CS(C)$:* Given a program's corpus C, the corpus population size is defined as the sum of all program's population sizes over the corpus C.

$$CS(C) = \Sigma S(P) \ \forall \ P \ \in \ C$$

Metric 4. *Corpus unique population size $UCS(C)$:* Given a program's corpus C, the corpus unique population size is defined as the sum of all program's unique population sizes over the corpus C

$$UCS(C) = \Sigma US(P) \ \forall \ P \ \in \ C$$

■ Protocol

To generate program variants, we synthesize program variants with an enumerative strategy, checking each synthesis for equivalence modulo input [**?**] against the original program, as it is described in **??**. For obvious reasons, this space is nearly impossible to explore in a reasonable time as soon as the limit of instructions increases. Therefore, we use two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. In our experiments for RQ1, we use all instructions in the CROW diversifier synthesis.

The former parameter allows us to find a trade-off between the number of variants that are synthesized and the time taken to produce them. For the current evaluation, given the size of the corpus and the properties of its programs, we set the exploration time to 1 hour maximum per function for Rosetta. In the cases of Libsodium and QrCode, we set the timeout to 5 minutes per function. The decision behind the usage of lower timeout for Libsodium and QrCode is motivated by the properties listed in **??**. The latter two corpora are remarkably larger regarding the number of instructions and functions count.

We pass each of the $303 + 869 + 1849$ functions in the corpora to CROW, as **??** illustrates, to synthesize program variants. We calculate the *Corpus population size*(**??**) and *Corpus unique population size*(**??**) for each corpus and conclude by answering RQ1.

■ 4.3 RQ_2. To what extent are the generated variants dynamically different?

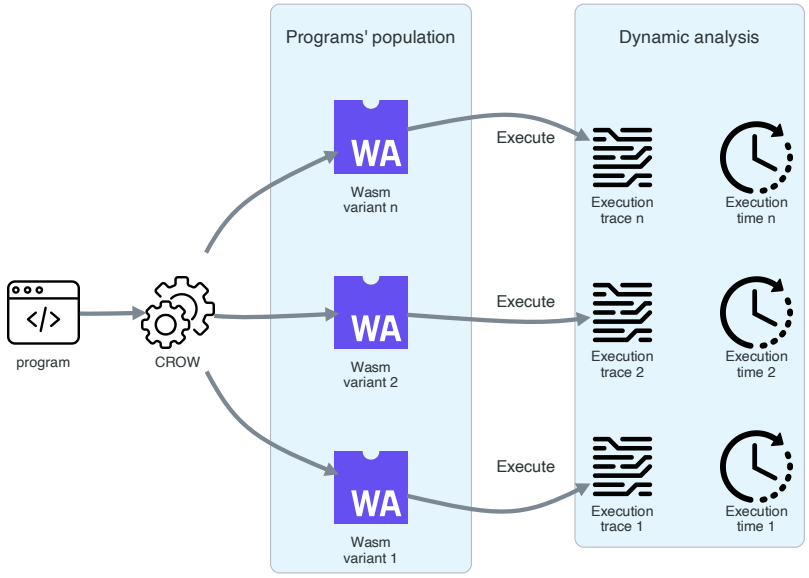


Figure 4.2: Dynamic analysis for RQ2.

In this second research question, we investigate to what extent the artificially created variants are dynamically different between them and the original program. To conduct this research question, we could separate our experiments into two fields as **??** illustrates: static analysis and dynamic analysis. The static analysis focuses on the appreciated differences among the program variants, as well as between the variants and the original program. We perform the static analysis in answering RQ1 in **??**. With RQ2, we focus on the last category, the dynamic analysis of the generated variants. This decision is supported because dynamic analysis complements RQ1 and, it is essential to provide a full understanding of diversification. We use the original functions from Rosetta corpus described in **??** and their variants generated to answer RQ1. We use only Rosetta to answer RQ2 because this corpus is composed of simple programs that can be executed directly without user interaction, *i.e.,* we only need to call the interpreter passing the WebAssembly binary to it. To dynamically compare programs and their variants,

we execute each program on each programs' population to collect and execution times. We define execution trace and execution time in the following section.

■ Metrics

We compare the execution traces of two any programs of the same population with a global alignment metric. We propose a global alignment approach using Dynamic Time Warping (DTW). Dynamic Time Warping [**?**] computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is a distance metric. The larger the DTW distance, the more different the two sequences are. DTW has been used for comparing traces in different domains. For software, De A. Maia et al. [**?**] proposed to identify similarity between programs from execution traces. As we discussed in **??**, a theoretical WebAssembly engine perform `push` and `pop` operations when the program instructions are executed. Therefore, in our experiments, we define the execution traces as the sequence of the stack operations during the execution of the WebAssembly program. In the following, we define the $TraceDiff$ metric.

Metric 5. *TraceDiff:* Given two programs P and P' from the same program's population, TraceDiff(P,P'), computes the DTW distance collected during their execution.
A TraceDiff of 0 means that both traces are identical. The higher the value, the more different the traces.

Moreover, we use the execution time distribution of the programs in the population to complement the answer to RQ2. For each program pair in the programs' population, we compare their execution time distributions. We define the execution time as follows:

Metric 6. *Execution time:* Given a WebAssembly program P, the execution time is the time spent to execute the binary.

■ Protocol

To compare program and variants behavior during runtime, we analyze all the unique program variants generated to answer RQ1 in a pairwise comparison using the value of aligning their execution traces (**??**). We use SWAM[2] to execute each program and variant to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions, including the virtual stack operations.

Furthermore, we collect the execution time, **??**, for all programs and their variants. We compare the collected execution time distributions between programs using a Mann-Withney U test [**?**] in a pairwise strategy.

[2] `https://github.com/satabin/swam`

■ 4.4 RQ_3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

TODO The last method is too short

To answer RQ3, we use the variants generated for the programs of Libsodium and QrCode corpora, we take $2 + 5$ programs interconnecting the LLVM bitcode modules (mentioned in **??**). We illustrate the protocol to answer RQ3 in **??** starting from the creation of the programs' population.

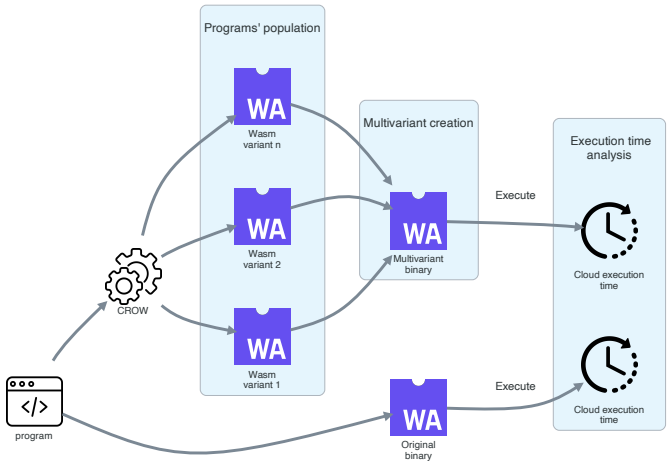


Figure 4.3: Multivariant binary creation and workflow for RQ3 answering.

In RQ3, we study whether the created variants can be used in real-world applications and what properties offer the composition of the variants as multivariant binaries. We build multivariant binaries (according to **??**), and we deploy and execute them at the Edge. The usage of Edge-Cloud computing platforms to answer RQ3 is motivated by two reasons. First, it is an emerging technology. Using Wasm as an intermediate layer is better in terms of startup and memory usage, than containerization or virtualization [**?** **?**]. This has encouraged edge computing platforms like Cloudflare and Fastly to use WebAssembly to deploy client applications in a modular and sandboxed manner [**?** **?**]. Second, Edge-Cloud computing platforms are shown to be not completely secure [**?**] and multivariant execution offers a preemptive technique against predictable behaviors such as execution time.

Metrics

To answer RQ3, we build multivariant WebAssembly binaries (see **??**) meant to provide execution path randomization. We use the execution time of the multivariant binaries to answer RQ3. We use the same metric defined in **??** for the execution time of multivariant binaries.

Protocol

We answer RQ3 by analyzing a real-world scenarios on the Edge. Edge applications are designed to be deployed as isolated HTTP services, having one single responsibility that is executed at every HTTP request. This development model is known as serverless computing, or function-as-a-service [**?** **?**]. We deploy and execute the multivariant binaries as end-to-end HTTP services on the Edge, and we collect their execution times. To remove the natural jitter in the network, the execution times are measured at the backend space, *i.e.,* we collect the execution times inside the Edge node and not from the client computer. Therefore, we instrument the binaries to return the execution time as an HTTP header.

We do the collection of the execution times twice, for the original program and its multivariant binary. We deploy and execute the original and the multivariant binaries on 64 edge nodes located around the world. In **??** we illustrate the word wide location of the edges nodes.

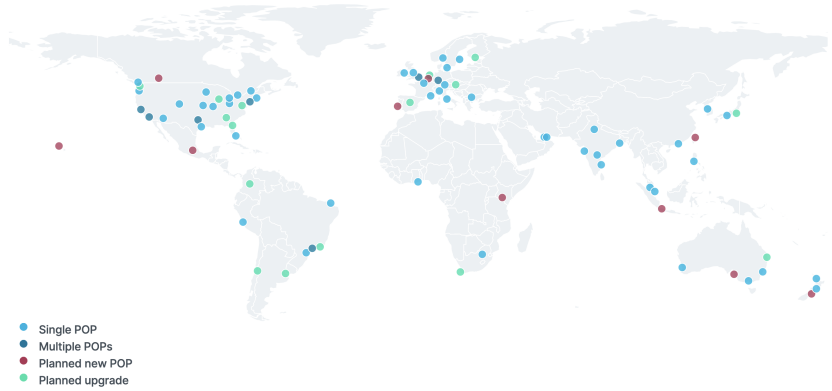


Figure 4.4: Screenshot taken from the Fastly Inc. platform used in our experiments for RQ3. Blue and darker blue dots represent the edge nodes used in our experiments.

We collect 100k execution times for each binary, both the original and multivariant binaries. The number of execution time samples is motivated by the seminal work of Morgan et al. [**?**]. We perform a Mann-Withney U test [**?**] to

compare both execution time distributions. If the P-value is lower than 0.05, the two compared distributions are different.

■ Conclusions

This chapter presents the methodology we follow to answer our three research questions. We first describe and propose the corpora of programs used in this work. We propose to measure the ability of our approach to generate variants out of 3021 functions of our corpora. Then, we suggest using the generated variants to study to what extent they offer different observable behavior through dynamic analysis. We propose a protocol to study the impact of the composition variants in a multivariant binary deployed at the Edge. Besides, we enumerate and enunciate the properties and metrics that might lead us to answer the impact of automatic diversification for WebAssembly programs. In the next chapter, we present and discuss the results obtained with this methodology.

05 RESULTS

In this chapter, we sum up the results of the research of this thesis. We illustrate the key insights and challenges faced in answering each research question. To obtain our results, we followed the methodology formulated in **??**.

5.1 RQ_1. To what extent can we artifically generate program variants for WebAssembly?

As we describe in **??**, our first research question aims to answer how to artifically generate WebAssembly program variants. This section is organized as follows. First we present the general results calculating the *Corpus population size*(**??**) and *Corpus unique population size*(**??**) for each corpus. Second, we discuss the challenges and limitations in program variants generation. Finally, we illustrate the most common code transformations performed by our approach and answer RQ1.

5.1.1 Program's populations

We summarize the results in **??**. The table illustrates the corpus name, the number of functions to diversify, the number of successfully diversified functions (functions with at least one artificially created variant), the cumulative number of variants (*Corpus population size*) and the cumulative number of unique variants (*Corpus unique population size*).

We produce at least one unique program variant for 239/303 single function programs for Rosetta with one hour for a diversification timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of Libsodium and QrCode, we produce variants for 85/869 and 32/1849 functions respectively, with 5 minutes per function as timeout. The rest of the functions resulted in timeout before finding function variants or produce no variants. For all programs in all corpora, we achieve 356/3021 successfully diversified functions, representing a 11.78% of the total. As the four and fifth columns show, the number of artifically created variants and the number of unique variants are larger than the original number of functions by one order of magnitude. In the case of Rosetta, the corpus population size is close to one million of programs. The remarkable difference between the total number of variants and the number

of unique variants (fourth and fifth columns) is mainly due to the *replacements combining* process discussed in **??**.

TODO M: add histogram on variant sizes

Corpus	#Functions	# Diversified	# Variants	# Unique Variants
Rosetta		239	809900	2678
Libsodium	869	85	4272	3805
QrCode	1849	32	6369	3314
	3021	356	820541	9797

Table 5.1: General program's populations statistics. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the cumulative number of generated variants and the cumulative number of unique variants.

5.1.2 Challenges for automatic diversification

We have observed a remarkable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions (third column of **??**). Our approach successfully diversified 239/303, 85/869 and 32/1849 of the original functions for Rosetta, Libsodium and QrCode respectively. The main reason of this phenomenon is the set timeout for CROW.

We have noticed a remarkable difference between the number of diversified functions for each corpus, 809900 programs for Rosetta 4272 for Libsodium and 6369 for QrCode. The corpus population size for Rosetta is two orders of magnitude larger compared to the other two corpora. The reason behind the large number of variants for Rosetta is that, after certain time, our approach starts to combine the code replacements to generate new variants. However, looking at the fifth column, the number of unique variants have the same order of magnitude for all corpora. The variants generated out of the combination of several code replacements are not necessarily unique. Some code replacements can dominate over others, generating the same WebAssembly programs.

A low timeout offers more unique variants compared to the population size despite the low number of diversified functions, like the Libsodium and QrCode cases. This happens because, CROW first generates variants out of single code replacements and then starts to combine them. Thus, more unique variants are generated in the very first moments of the diversification process with CROW.

Apart from the timeout and the combination of variants phenomenon, we manually analyze programs, searching for properties attempting to the generation of program variants using CROW. As we previously mentioned in **??**, *constant*

inferring is a new contribution of ours to the collection of Software Diversification strategies enumerated in **??**. We have observed that our approach searches for a constant inferring for more than 45% of the instructions of each function while constant values cannot be inferred in all cases. The main reason is that memory operations are also included into the inferring while our tool is oblivious to a memory model, making unsuccessful the constant replacement.

5.1.3 Properties for large diversification

We manually analyzed the programs to study the critical properties of programs producing a high number of variants. This reveals one key factor that favors many unique variants: the presence of bounded loops. In these cases, we synthesize variants for the loops by replacing them with a constant, if the constant inferring is successful. Every time a loop constant is inferred, the loop body is replaced by a single instruction. This creates a new, statically different program. The number of variants grows exponentially if the function contains nested loops for which we can successfully infer constants.

A second key factor for synthesizing many variants relates to the presence of arithmetic. The synthesis engine used by our approach, effectively replaces arithmetic instructions with equivalent instructions that lead to the same result. For example, we generate unique variants by replacing multiplications with additions or shift left instructions (**??**). Also, logical comparisons are replaced, inverting the operation and the operands (**??**). Besides, our implementation can use overflow and underflow of integers to produce variants (**??**), using the intrinsics of the underlying computation model.

Listing 5.1: Diversification through arithmetic expression replacement.

```
local.get 0
i32.const 2
i32.mul
```

```
local.get 0
i32.const 1
i32.shl
```

Listing 5.2: Diversification through inversion of comparison operations.

```
local.get 0
i32.const 10
i32.gt_s
```

```
i32.const 11
local.get 0
i32.le_s
```

Listing 5.3: Diversification through overflow of integer operands.

```
i32.const 2
i32.mul
```

```
i32.const 2
i32.mul
i32.const
  -2147483647
i32.mul
```

At the WebAssembly level, we have not observed variants performing changes in the control flow structure of the variants (**??**). We manually analyze the machine code generated by V8 (as it was discussed in **??**). We have observed that, for different variants, we are changing the number of jumps and its location inside the machine code.

Answer to RQ1.

We can provide diversification for 11.78% of the programs in our corpora. Constant inferring, complemented with the high presence of arithmetic operations and bounded loops in the original program increased the number of program variants.

5.2 RQ_2. To what extent are the generated variants dynamically different?

Our second research question investigates the differences between program variants at runtime. To answer RQ2, we execute each program/variant generated to answer RQ1 for Rosetta corpus to collect their execution traces and execution times. For each programs' population we compare the stack operation traces (**??**) and the execution time distributions (**??**) for each program/variant pair.

This section is organized as follows. First, we analyze the programs' populations by comparing the traces for each pair of program/variant with TraceDiff of **??**. The pairwise comparison will hint at the results at the population level. We analyze not only the differences of a variant regarding its original program, we also compare the variants against other variants. Second, we do the same pairwise strategy for the execution time distributions **??**, performing a Mann-Withney U test for each pair of program/variant times distribution. Finally, we conclude and answer RQ2.

5.2.1 Stack operation traces.

In **??** we plot the distribution of all comparisons (in logarithmic scale) of all pairs of program/variant in each programs' population. All compared programs are statically different. Each vertical group of blue dots represents all the pairwise comparison of the traces (**??**) for a program of Rosetta corpus for which we generate variants. Each dot represents a comparison between two programs' traces according to **??**. The programs are sorted by their number of variants in descending order. For the sake of illustration, we filter out those programs for which we generate only 2 unique variants.

We have observed that in the majority of the cases, the mean of the comparison values is remarkably large. We analyze the length of the traces, and one reason behind such large values of TraceDiff is that some variants result from constant inferring. For example, if a loop is replaced by a constant, instead of several symbols in the stack operation trace, we observe one. Consequently, the distance between two program traces is significant.

In some cases, we have observed variants that are statically different for which TraceDiff value is zero, *i.e.,* they result in the same stack operation trace. We

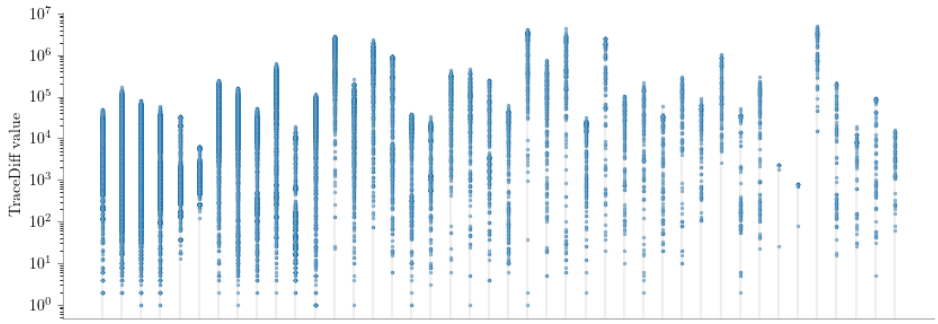


Figure 5.1: Pairwise comparison of programs' population traces in logarithmic scale. Each vertical group of blue dots represents a programs' population. Each dot represents a comparison between two program execution traces according to **??**.

identified two main reasons behind this phenomenon. First, the code transformation that generates the variant targets a non-executed or dead code. Second, some variants have two different instructions that trigger the same stack operations. For example, the code replacements below illustrate the case.

(1) `i32.lt_u`	`i32.lt_s`	(3) `i32.ne`	`i32.lt_u`
(2) `i32.le_s`	`i32.lt_u`	(4) `local.get 6`	`local.get 4`

In the four cases, the operators are different (original in gray color and the replacement in green color) leaving the same values for equal operands. The (1) and (2) cases are comparison operations leaving the value 0 or 1 in the stack taking into account the sign of their operands. In the third case, the replacement is less restricted to the original operator, but in both cases, the codes leave the same value in the stack. In the last case, both operands load a value of a local variable in the stack, the index of the local variable is different but the value of the variable that is appended to the trace is the same in both cases.

■ 5.2.2 Execution times.

Even when two programs of the same population offer different execution traces, their execution times can be similar (statistically speaking). In practice, the execution traces of WebAssembly programs are not necessarily accessible, being not the case with the execution time. For example, in our current experimentation we need to use our own instrumentation of the execution engine to collect the stack trace operations while the execution time is naturally accessible in any execution environment. This mentioned reasoning enforces our comparison of the execution times for the generated variants. Besides the execution times of programs can be

used by malicious clients to construct personalized attacks [**?**]. Therefore, by measuring the execution times, we assess the diversification of observable behaviors evaluated in real-world security scenarios.

For each program's population, we compare the execution time distributions, **??**, of each pair of program/variant. Overall diversified programs, 169 out of 239 (71%) have at least one variant with a different execution time distribution than the original program (P-value < 0.01 in the Mann-Withney test). This result shows that we effectively generate variants that yield significantly different execution times.

By analyzing the data, we observe the following trends. First, if our tool infers control-flows as constants in the original program, the variants execute faster than the original, sometimes by one order of magnitude. On the other hand, if the code is augmented with more instructions, the variants tend to run slower than the original.

In both cases, we generate a variant with a different execution time than the original. Both cases are good from a randomization perspective since this minimizes the certainty a malicious user can have about the program's behavior. Therefore, a deeper analysis of how this phenomenon can be used to enforce security will be discussed in answering RQ3.

To better illustrate the differences between executions times in the variants, we dissect the execution time distributions for one programs' population of Rosetta. The plots in **??** show the execution time distributions for the `Hilbert curve` program and their variants. We illustrate time diversification with this program because, we generate unique variants with all types of transformations previously discussed in **??**. In the plots along the X-axis, each vertical set of points represents the distribution of 100000 execution times per program/variant. The Y-axis represents the execution time value in milliseconds. The original program is highlighted in green color: the distribution of 10000 execution times is given on the left-most part of the plot, and its median execution time is represented as a horizontal dashed line. The median execution time is represented as a blue dot for each execution time distribution, and the vertical gray lines represent the entire distribution. The bolder gray line represents the 75% interquartile. The program variants are sorted concerning the median execution time in descending order.

For the illustrated program, many diversified variants are optimizations (blue dots below the green bar). The plot is graphically clear, and the last third represents faster variants resulting from code transformations that optimize the original program. Our tool provides program variants in the whole spectrum of time executions, lower and faster variants than the original program. The developer is in charge of deciding between taking all variants or only the ones providing the same or less execution time for the sake of performance. Nevertheless, this result calls for using this timing spectrum phenomenon to provide binaries with unpredictable execution times by combining variants. The feasibility of this idea will be discussed in **??**.

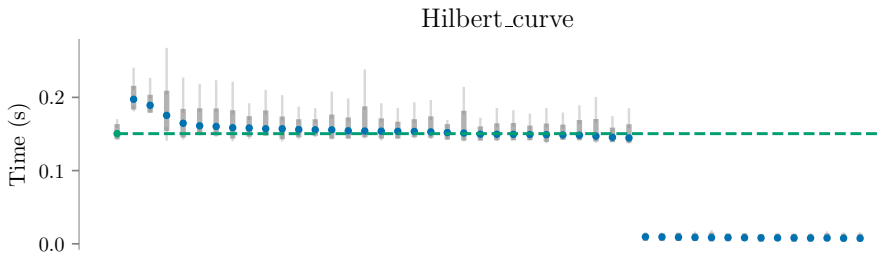


Figure 5.2: Execution time distributions for `Hilber_curve` program and its variants. Baseline execution time mean is highlighted with the magenta horizontal line.

Answer to RQ2.

We empirically demonstrate that our approach generates program variants for which execution traces are different. We stress the importance of complementing static and dynamic studies of programs variants. For example, if two programs are statically different, that does not necessarily mean different runtime behavior. There is at least one generated variant for all executed programs that provides a different execution trace. We generate variants that exhibit a significant diversity of execution times. Concretely, for 169/239 (71%) of the diversified programs, at least one variant has an execution time distribution that is different compared to the execution time distribution of the original program. The result from this study encourages the composition of the variants to provide a resilient execution.

■ 5.3 RQ_3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

Here we investigate the impact of the composition of program variants into multivariant binaries. To answer this research question, we create multivariant binaries from the program variants generated for Libsodium and QrCode corpora. Then, we deploy the multivariant binaries into the Edge and collect their execution times.

5.3.1 Execution times

We compare the execution time distributions for each program for the original and the multivariant binary. All distributions are measured on 100k executions of the program along all Edge platform nodes. We have observed that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In **??**, each subplot represents the quantile-quantile plot [**?**] of the two distributions, original and multivariant binary. This kind of plots is used to compare the shapes of distributions, providing a graphical comparison of location, scale, and skewness for two distributions. The dashed line cutting the subplot represents the case in which the two distributions are equal, *i.e.,* for two equal distribution we would have all blue dots over the dashed line. These plots reveal that the execution times are different and are spread over a more extensive range of values than the original binary. The standard deviation of the execution time values evidences the latter, the original binaries have lower values while the multivariant binaries have higher values up to 100 times the original. Besides, this can be graphically appreciated in the plots when the blue dots cross the reference line from the bottom of the dashed line to the top. This is evidence that execution time is less predictable for multivariant binaries than original ones. This phenomenon is present because the choice of function variants is randomized at each function invocation, and the variants have different execution times due to the code transformations, i.e., some variants execute more instructions than others.

Answer to RQ3.

The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. The distribution for the multivariant binary is different and even more spread than the original one. Consequently, attacks relying on measuring precise execution times [**?**] of a function are made a lot harder to conduct.

Conclusions

Our approach introduces static and dynamic, variants for up to 11.78% of the programs in our three corpora, increasing the original count of programs by 4.15 times. We highlighted the importance of complementing static and dynamic studies for programs diversification. Moreover, combining function variants in multivariant binaries makes virtually impossible to predict which variant is executed for a

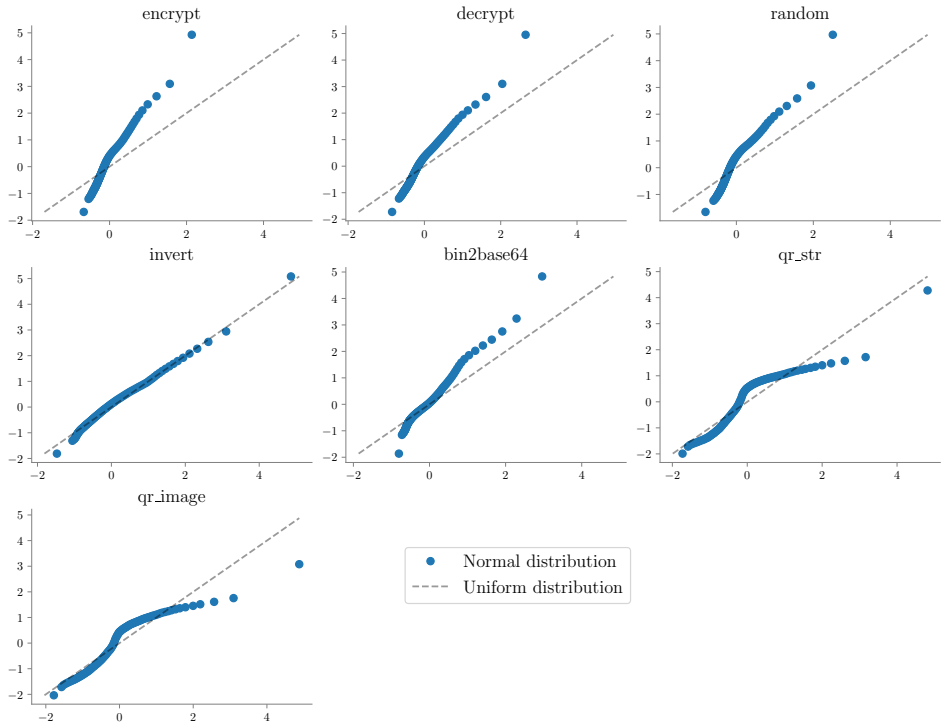


Figure 5.3: Execution time distributions. Each subplot represents the quantile-quantile plot of the two distributions, original and multivariant binary.

given query. We empirically demonstrate the feasibility and the application of automatically generating WebAssembly program variants.

06 CONCLUSION AND FUTURE WORK

WebAssembly has become a new technology for web browsers and standalone engines such as the ones used in Edge-Cloud platforms. WebAssembly is designed with security and sandboxing premises, yet, is still vulnerable. Besides, since it is a relatively new technology, new vulnerabilities appear in the wild faster than the adoption of patches and defenses. As a widely studied field, software diversification could be a solution for known and yet-unknown vulnerabilities. Yet, there is no research on this field for WebAssembly.

We propose an automatic approach to generate software diversification for WebAssembly in this work. In addition, we provide complementary implementation for our approaches, including a generic LLVM superdiversifier that potentially extends our ideas to other programming languages. We empirically demonstrate the impact of our approach by providing Randomization and Multivariant Execution (MVE) for WebAssembly. For this, we provide two tools, CROW and MEWE. CROW completely automatizes the process by using a superdiversifier. Besides, MEWE provides execution path randomization for an MVE. This chapter is organized into two sections. In **??**, we summarize the main results we found by answering our research questions enunciated in **??**. Finally, **??** describes potential future work that could extend this dissertation.

6.1 Summary of the results

We enunciate the three research questions in **??**. With the first research question, we investigate the static properties of the software diversification for WebAssembly generated by our approaches. We answer our first research question by creating programs variants for 3021 original programs. With CROW, we create program variants for the 11.78% of the programs in our corpora. We study the properties of the generated variants at the level of generated programs' population. Thus, we identify the challenges that attempt against the generation of unique program variants. Besides, we highlight the code properties that offer numerous program variants.

Complementary with our first research question, we evaluate the dynamic properties of the program variants generated to answer our first research question. We execute each of the 303 original programs and its generated variants for the Rosetta. For each execution, we collect their execution trace and their execution times. We demonstrate that the WebAssembly variants generated by CROW offer

remarkably different execution traces. Similarly, the execution times are different between each program and its variants. For the 71% of the diversified programs, at least one variant has an execution time distribution that is different from the original program's execution time distribution. Moreover, CROW generates both faster and slower variants. Nevertheless, we highlighted the importance of dynamic analysis for software diversification.

Our last and third research question evaluates the impact of providing a worldwide MVE for WebAssembly. We use MEWE to build multivariant binaries for the program variants generated for Libsodium and QrCode corpora. We deploy the generated multivariant binaries in an Edge-Cloud platform, collecting their execution times. The addition of runtime path randomization to multivariant binaries provides significant differences between the execution of the original binary and the multivariant binary. The observed differences lead us to conclude that no specific variant can be inferred from studying the execution time of the multivariant binaries. Therefore, attacks that rely on measuring precise execution times are more challenging to conduct.

Overall, these results show that our approaches can provide an automated end-to-end solution for the diversification of WebAssembly programs. Our approaches harden observable properties commonly used to conduct attacks, such as static code analysis, execution traces, and execution time. Therefore, our approaches harden unknown and yet-unknown vulnerabilities.

■ 6.2 Future work

Along with this dissertation, we highlighted challenges and limitations. In all cases, we proposed solutions, yet, some of them could be explored more in-depth as a call for optimization. For example, as we mentioned in **??** our solution provides program variants but remarkably lower unique variants as a consequence of the replacement combining process of CROW (**??**). Techniques relying on intelligent heuristics could help increase the generation of unique variants by early discarding unsound combinations. On the other hand, constant inferring does not always finish in a successful replacement due to the CROW's obliviousness to some computation models, such as memory operations. A solution could also be to use heuristics to select which part of the code is more probable to become a constant assignment.

As we mentioned in **??**, another approach to providing software diversification for Wasm could be binary to binary transformations. This approach could be used to increase resilience in malware classifiers through the study of diversification as an obfuscation technique [**?**]. Obfuscation of Wasm code could be used to measure the accuracy of malware classifiers.

By using a superdiversifier, the generated code transformations outperform hand-written transformations. Evidence of this is the CVE[1] found in the code

[1] `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`

generation component of wasmtime. We found this CVE during the implementation of MEWE with one of the generated variants. This highlighted the need for better strategies for stressing compilers, interpreters, and validators of Wasm. CROW and MEWE might improve fuzzing campaigns [**?**], preventing vulnerabilities by providing better testing.

Part II

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Programming 2020, MoreVMs'20

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
NDSS 2021, MADWeb

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Under review

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
SPLASH 2019, VMIL